SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 3–33 (1996)

Research

DECODE: A Co-operative Program Understanding Environment

DAVID N. CHIN

Department of Information and Computer Science, University of Hawaii, 2565 The Mall, Honolulu, HI 96822, U.S.A.

ALEX QUILICI

Department of Electrical Engineering, University of Hawaii, 2540 Dole Street, Holmes 483, Honolulu, HI 96822, U.S.A.

SUMMARY

The large size and high-percentage of domain-specific code in most legacy systems makes it unlikely that automated tools will be able to extract a complete underlying design. Yet automated tools can clearly recognize portions of the design. This suggests exploring environments in which programmer and system work together to understand legacy software. DECODE is such an environment. It supports programmer and system co-operation to extract design information from legacy software systems. DECODE's automated program understanding component recognizes standard implementations of domain independent plans to produce an initial knowledge base of object-orientated design elements. DECODE's structured notebook component provides the user with a graphical view of the initial understanding, which the user can extend by linking arbitrary source code fragments to either existing or new design elements, and then uses this design information to support conceptual queries about the program's code and design.

KEY WORDS: program understanding; reverse engineering, co-operative environment; object-orientated; design extraction

1. INTRODUCTION

Conceptual design information about legacy software is needed for tasks such as general software maintenance, porting the software to new architectures and/or languages, and identifying reusable components. All too often, however, software engineers must extract this information directly from the source code. This undesirable situation arises because the available documentation is either insufficient or incorrect and because the software's original designers are either unavailable or do not remember the details of its design.

Many researchers have explored automated mechanisms for performing this design extraction. The standard approach is to try to recognize the instances of a library of known code patterns (also called programming plans, clichés, and so on) (Kozaczynski and Ning, 1994; Kozaczynski, Wing and Engberts, 1992; Wills, 1990, 1992; Hartman, 1991; Rich and Waters,

CCC 1040–550X/96/010003–31

Received 15 October 1994
Revised 28 September 1995

1990; Harandi and Ning, 1990; Letovsky, 1988; Murray, 1988; Johnson, 1986). Unfortunately, there are several fundamental problems with trying to apply this approach to large, real-world legacy systems:

1. *This approach requires enormous libraries of code patterns.* Every domain has its own domain-specific design elements, each of which requires a set of domain-specific code patterns to represent its different implementations.
2. *Current program-understanding algorithms are not applicable to legacy software systems.* Bottom-up approaches tend not to scale well with the size of the program or the size of the pattern library, while top-down approaches tend to require often unavailable detailed advanced knowledge about what the program is doing.
3. *Some code simply does not fit into patterns.* There are always novel applications and domains for which no code pattern library will yet exist. Beyond that, however, programming is by its very nature idiosyncratic (Detienne and Soloway, 1990; Letovsky and Soloway, 1986; Soloway and Erdlich, 1984).

The bottom line is that even if we could form a large code pattern library that spanned a variety of domains, current program understanding algorithms would no longer be applicable. And even if they were, there would still be code that was not covered by existing code patterns. As a result, any understanding system cannot be expected to automatically extract a program's entire design. Instead, software understanding must be viewed as a co-operative process involving both programmer and system, where at a minimum the system extracts what design knowledge it can automatically and then assists programmers in augmenting this understanding.

This viewpoint has led us to construct DECODE (Diagram-based Environment for Co-operative Object-orientated Design Extraction), a prototype program understanding environment in which programmers and system co-operate to extract designs from legacy COBOL programs. In particular, DECODE provides mechanisms for:

1. Automatically extracting information about stereotypical portions of a COBOL program's design.
2. Helping programmers visually create a knowledge base containing a detailed design of an existing software system.
3. Using this programmer and system-created knowledge base to answer *conceptual* queries about this software system.

These queries are conceptual in the sense that they cannot be answered by simple text-based searches, nor can they be answered by referring to structural relationships between program entities. Instead, they can only be answered by accessing the program's jointly extracted design. One example of a conceptual query is 'where does the program perform "input variation"?', a query that might be asked by a maintenance programmer trying to extend an existing program to handle a new type of input record. Another example is 'where are error messages constructed and written?', which might be asked by a programmer trying to modify the output format of existing error messages.

Conceptual queries differ significantly from *structural* queries, such as 'what functions call *F*?' or 'what code is dependent on variable *V*'s value?'. Many research efforts have been devoted to automatically forming the databases describing a program's structure that are

necessary to answer these queries, such as data flow diagrams, call graphs and other dependencies (Takeda, Chin and Miyamoto, 1992; Chen, Nishimoto and Ramamoorthy, 1990; Wilde, Huitt and Huitt, 1989; Kuhu, 1987), and to making this knowledge easy to access (Selfridge and Heineman, 1994; Ball and Eick, 1994). While this structural knowledge is often straightforward to extract automatically, its usefulness is limited to answering queries about structural relationships. In contrast, answering conceptual queries requires knowledge about design relationships that cannot always be extracted automatically, which forces us to consider how the programmer and system can work together to extract it.

This article describes DECODE and is organized as follows. Section 2 presents a brief overview of the system's components and capabilities. Section 3 describes the system's approach to automated design extraction. Section 4 describes the system's structured notebook for assisted understanding and querying of the extracted design. Section 5 describes the paradigms of DECODE's actual use to understand programs. Section 6 describes systems that are closely related to DECODE and discusses their relative strengths and weaknesses. Section 7 describes DECODE's most important shortcomings and how we plan to address them. Section 8 describes the conclusions we have drawn from constructing and initially using DECODE.

2. SYSTEM OVERVIEW

Figure 1 is an overview of DECODE's architecture. It has three components: an automated programming plan recognizer, a knowledge base for recording extracted design information, and a structured notebook for editing and querying this design. It assumes that this design should be recorded in an object-orientated style (i.e., in terms of a collection of objects and operations on them).

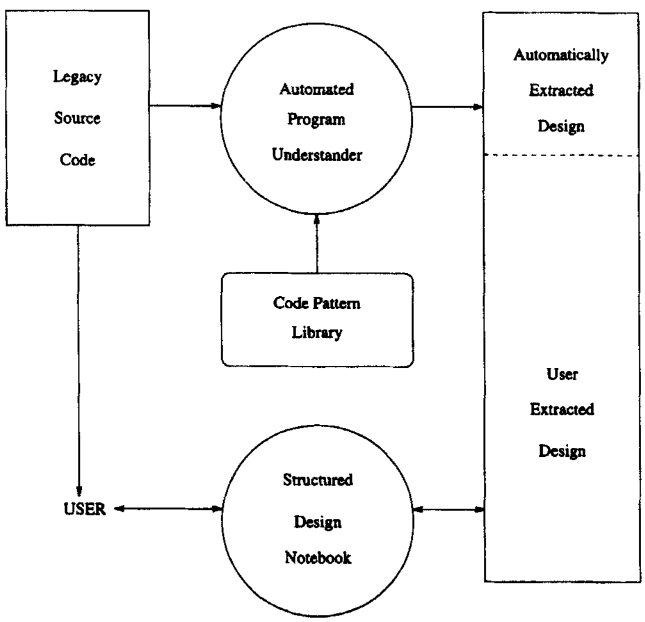

Figure 1. DECODE's architecture

DECODE takes an existing program and forms a partial understanding of it by examining it for instances of standard code patterns. This forms the initial knowledge base. The programmer then inspects this program's source code and tries to extend this partial understanding. The programmer does this by using the structured notebook, which supports three main operations (all of which are carried out visually): creating design primitives (including conceptual objects, operations, and the relationships between them), examining and selecting code fragments and linking them to design primitives, and asking conceptual queries of the code and its relationship to the extracted design. Essentially, whenever the programmer recognizes the purpose of some arbitrary code segment, the programmer selects this code and maps it to an object or operation in a programmer-created object-orientated hierarchy.

DECODE's current focus is to help the user extract and record one key portion of a program's complete object-orientated design. In particular, it concentrates on static design information: conceptual objects, operations, and the various relationships between these objects and operations. A complete object-orientated design, however, consists of information about the dynamic behaviour of the program as well. Thus, DECODE cannot be used to extract a complete object-orientated design of a system that could be used to produce code. Instead, DECODE is designed to help users understand and record the portion of the program's design that revolves around data and the ways they are used (an approach to understanding legacy systems that has been shown to be useful by other efforts, such as Zimmer (1990)). This information can then be used as a higher-level starting point for extracting other portions of the complete design.

3. THE AUTOMATED DESIGN EXTRACTOR

The design extractor attempts to recognize programming plans and the conceptual design elements they implement. These include both simple plans (whose components correspond to abstract syntax tree entries) and compound plans (whose components may themselves be plans).

Automated design extraction—an example

Figure 2 illustrates the design extractor's capabilities by showing a portion of the design that is automatically extracted from the COBOL program in the Appendix. This includes the highest-level plans it recognized and the conceptual design elements (classes and operations) they implement.

One automatically recognized simple plan is DISPLAY-LABELLED-RECORD (the plan of writing a message followed by the record). There are eight instances of this plan in our example program. One instance is:

```
140 MOVE SPACES TO PRINT-RECORD
141 MOVE 'INVALID TRANSACTION' TO PRT-MESSAGE
142 MOVE TRANSACTION-RECORD TO PRT-REC
143 WRITE PRINT-RECORD
```

Another instance of this same plan is:

```
 98 MOVE SPACES TO PRINT-RECORD
 99 MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE
397 MOVE NEW-RESERVATIONS-RECORD TO PRT-REC
398 WRITE PRINT-RECORD
```

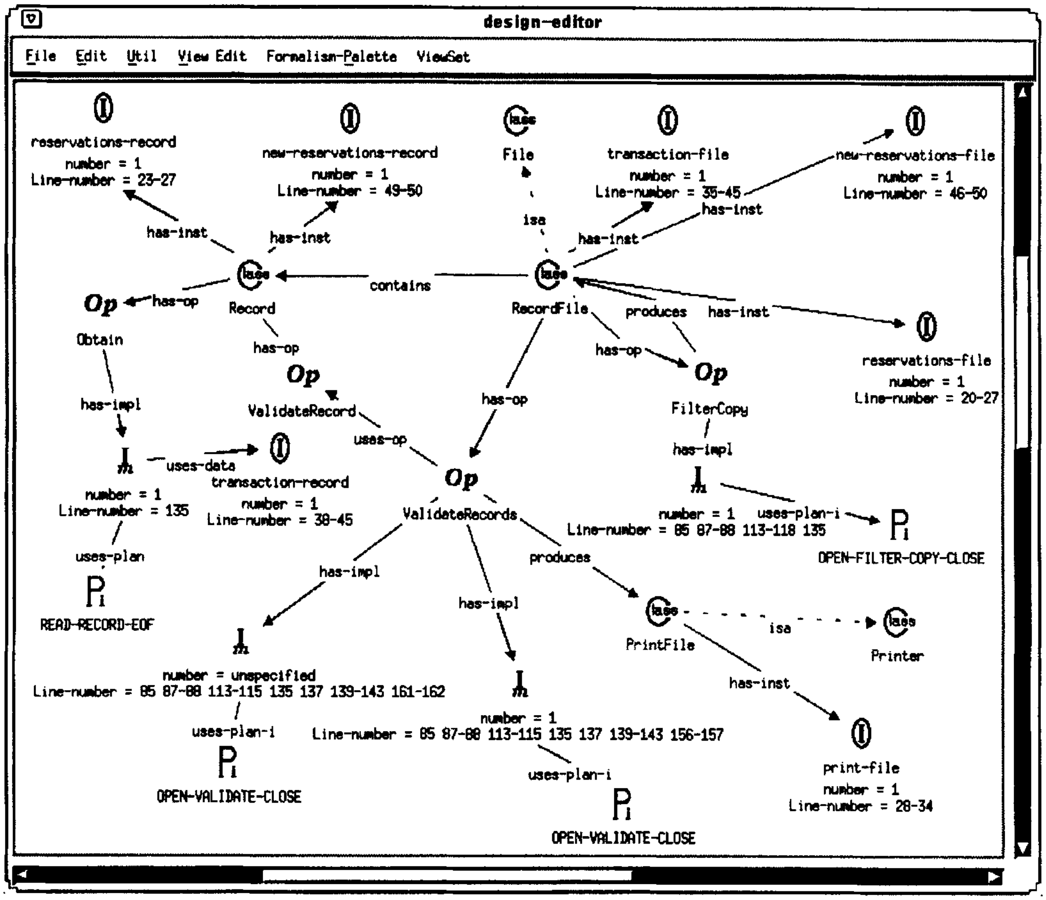

Figure 2. A portion of the raw design that DECODE is capable of automatically recognizing. To aid readability, the APU initially displays only the highest-level plans it recognizes

Given our current plan library, our design extractor recognizes a set of simple plans in this program, which include READ-RECORD-EOF (the plan of reading a record and setting a flag to record that end of file was encountered) and REMEMBER-CONDITION (the plan of setting a flag to record that a particular condition holds).

In addition to simple plans like these, the design extractor also recognizes instances of compound plans, such as OPEN-VALIDATE-CLOSE (the plan of reading all input file records and printing a message identifying each record that fails a particular test) and OPEN-FILTER-COPY-CLOSE (the plan of reading all input file records and copying into another file those meeting a particular test). Both are recognized from other compound plans, such as VALIDATE-ALL-RECORDS (the plan of reading and validating input records), which are in turn recognized from the simpler plans shown above.

Finally, the design extractor automatically links instances of certain code patterns to various design elements. For example, it links instances of the plan OPEN-VALIDATE-CLOSE to the ValidateRecords operation on objects in the class RecordFile. Similarly, it links the plan OPEN-FILTER-COPY-CLOSE to the FilterCopy operation on Record Files. As a

result, the design extractor can work its way up from the code to a portion of the high-level design that describes the program.

Practical uses of partial design extraction

Even partial automatic recognition of the programming plans is a useful aid to maintenance programmers. In particular, it can be used to locate code segments relevant to a proposed change. For example, consider a maintenance programmer charged with modifying our example program's output to indicate errors explicitly (e.g., by modifying the label on error messages to contain the string 'Error:'). To know which lines to modify, the programmer need only request that the system display all recognized instances of DISPLAY-LABELLED-RECORD. Without this automatic recognition, however, the programmer would likely examine every WRITE and determine whether it was preceded by the appropriate MOVEs—a difficult task, as any plan's components can be widely distributed within the source code. The second instance of DISPLAY-LABELLED-RECORD above, for example, is spread out across the program (moving spaces and storing the label into the print record occur near the beginning, while completing the record and writing it occurs near the end).

The plan library cannot be guaranteed to be complete, so the design extractor may not recognize all instances of a particular plan. As a result, the programmer above must still examine the program for alternative implementations of DISPLAY-LABELLED-RECORD. This task, however, is simpler than starting from scratch, as the programmer can focus on only those WRITEs not recognized as part of a DISPLAY-LABELLED-RECORD.

The automatic recognition of higher-level design elements also supports locating the portions of the program relevant to proposed changes. For example, consider a maintenance programmer trying to determine which conditions a program is validating in its input transactions (e.g., to track down a bug where a particular condition is not being checked correctly). By requesting all instances of the high-level design element, ValidateRecords, the programmer can see at a glance what part of the program is concerned with input validation. That greatly narrows down the search for the particular conditions the program is actually checking. Again, as with programming plans, the design extractor is not guaranteed to recognize all instances of a particular high-level design element, so the maintenance programmer will still need to examine the remainder of the program by hand.

Our approach to design extraction

Our design extractor is based on earlier program understanding research, especially work dealing with specific problems such as code variability and in finding algorithms for locating patterns in the code (Kozaczynski and Ning; 1994; Kozaczynski, Ning and Engberts, 1992; Wills, 1990, 1992; Hartman, 1991; Letovsky, 1988; Murray, 1988; Johnson, 1986). A key difference, however, is that all these efforts assume that the library of code patterns and the resulting design description are static. In contrast, DECODE tries to relax both of these assumptions. DECODE's design extractor addresses the problems inherent in large code pattern libraries by using an opportunistic recognition algorithm and a highly organized pattern library. It addresses the problems of users updating the resulting design description by explicitly trying to connect plans to higher-level design elements and by allowing users the same graphical access to recognized plans as to higher-level design elements.

This design extractor is described in detail in (Quilici, 1994) and is derived from studies

of users doing bottom-up understanding on functions in C code (Quilici, 1993). Below we summarize the crucial aspects of our approach and discuss their effectiveness.

Background: the Concept Recognizer

Our design extractor is based on the Concept Recognizer (Kozaczynski and Ning, 1994; Kozaczynski, Ning and Engberts, 1992). The Concept Recognizer divies plans into two parts: a description of the plan's attributes (which are instantiated when a plan instance is recognized) and a set of common implementation patterns. It represents these code patterns as a combination of *components* (the particular language items or subplans that must be recognized to have a potential instance of the plan) and *constraints* (the relationships that must hold between these components).

Figure 3 illustrates how the plan DISPLAY-LABELLED-RECORD would be represented by the Concept Recognizer.

The plan falls into the general class of DISPLAY-ITEM plans and has two attributes to be filled-in whenever an instance of this plan is recognized (the Record-Name and the Message). One implementation of this plan consists of four components: a FILL-WITH-SPACES sub-plan to clear the output record, two MOVEs to place the message and the data to display into the output record, and a WRITE to display the output record. However, not any combination of a FILL-WITH-SPACES, two MOVEs and a WRITE is an instance of the plan. There also must be a data dependency between the WRITE and the MOVEs that transfer data into the output record. The output record must be cleared at some point before these MOVEs, and the destinations for the MOVEs must be fields within the output record. Only if all these constraints hold do we have an instance of DISPLAY-LABELLED-RECORD.

The Concept Recognizer takes a library-driven approach to plan recognition. It takes each code pattern in the library, matches its components against the program, and then applies constraints to the set of candidate plans (actually, it tries to interleave constraint checking and matching). When a component can itself be a plan, the algorithm recursively tries to recognize instances of that plan. While its representation of plans is both simple and clear and its algorithm is successful at recognizing plans in real-world COBOL programs, the algorithm is slow and does not scale well, either with program size or plan library size (Kozaczynski and Ning, 1994).

```
plan DISPLAY-LABELLED-RECORD(Record-Name, Message) isa DISPLAY-PLAN

recognize DISPLAY-LABELLED-RECORD(Record-Name: ?rec, Message: ?msg)
  components
    Clear-Record: FILL-WITH-SPACES(Dest: ?pr)
    Provide-Msg:  MOVE(Source: ?msg, Dest: ?msg-field)
    Provide-Rec:  MOVE(Source: ?rec, Dest: ?rec-field)
    Dump-Record:  WRITE(Source: ?pr)
  constraints
    DataDep(Write-Record, Provide-Msg, ?msg-field)
    DataDep(Write-Record, Provide-Rec, ?rec-field)
    ControlPath(Clear-Record, Provide-Msg)
    ControlPath(Clear-Record, Provide-Rec)
    Field(?msg-field, ?pr)
    Field(?rec-field, ?pr)
```

Figure 3. An example of the Concept Recognizer's representation of code patterns

Our modified Concept Recognizer

Our design extractor differs from the Concept Recognizer in three significant ways. First, it is code-driven (bottom-up) rather than library-driven (top-down). While library-driven approaches consider all plans in the library, code-driven approaches consider only the subset of those plans that contain already-recognized components. Second, we use careful indexing and organization of the plan library to attempt to further reduce the plans that are considered, the number of constraints that must be evaluated, and the amount of matching that must take place between the code and the plan library. Third, we have made several modifications that simplify the process of providing the plans in the plan library. Figure 4 contains several examples of DECODE's extended plan representation.

Indexing

Each plan in DECODE's library has an index that says when it should be considered (that is, fully matched against known program pieces and recognized plans). The index combines a plan component with one or more plan constraints and suggests that the plan should be considered whenever this component is encountered and the specified constraints hold. DISPLAY-LABELLED-RECORD, for example, is indexed by a WRITE that is data dependent on a MOVE to one of the fields in the record being written. This means that the design extractor considers this plan only when it encounters a WRITE, not every time it encounters a MOVE or a WRITE (as in most bottom-up approaches). Evaluating the index involves trying to verify the constraints hold (which may in turn involve trying to match additional plan components). In this case, it involves determining what fields are contained within the record being written and locating the MOVEs involving those fields on which the WRITE is data dependent.

The idea is that indexes suggest when plans are *likely* to occur as opposed to when plans *might* occur. This has the potential for several beneficial effects. First, it may cut down on the number of plans in the library that are considered during understanding, as any plan that is not indexed by the elements of a given program will never be considered. Although over half of the plans in our plan library involve MOVEs, almost all these plans (except for EXCHANGE-VALUES) can be indexed by other elements, allowing indexing to cut the number of unique plans considered by approximately half.

Second, indexing has the potential to significantly reduce the number of times any given plan is considered by a bottom-up understander. As an illustration, our example program contains 41 MOVEs, 10 FILL-WITH-SPACESs, and nine WRITEs, where each MOVE can appear in two different places within the DISPLAY-LABELLED-RECORD plan. The result is that a straightforward bottom-up plan recognizer must consider 102 initial partial plan instances (10 FILL-WITH-SPACES for the Clear-Record component, 41 MOVEs for the Provide-Msg component, 41 MOVEs for the Provide-Rec component, and 10 WRITEs for the Dump component), where each partial plan instance is an initial binding of DISPLAY-LABELLED-RECORD based on the match of a component with a program piece. However, our indexed bottom-up approach need only initially consider nine possible plan instances (one for each WRITE) and furthermore need only subsequently consider those MOVEs that are related to those WRITEs by a data dependency involving the record the WRITEs are displaying.

Finally, indexing has the potential to reduce the amount of matching and constraint evaluation that takes place while recognizing instances of a particular plan. Ideally, the recognition

```
DISPLAY-LABELLED-RECORD(Record-Name: ?rec, Message: ?msg)
  components
    Clear-Record:  FILL-WITH-SPACES(Dest: ?pr)
    Provide-Msg:   MOVE(Source: ?msg, Dest: ?msg-field)
    Provide-Rec:   MOVE(Source: ?rec, Dest: ?rec-field)
    Dump-Record:   WRITE(Source: ?pr)
  constraints
    Msg-N-Dump:    DataDep(Write-Record, Provide-Msg, ?msg-field)
    Rec-N-Dump:    DataDep(Write-Record, Provide-Rec, ?rec-field)
    Clr-N-Msg:     ControlPath(Clear-Record, Provide-Msg)
    Clr-N-Rec:     ControlPath(Clear-Record, Provide-Rec)
    Msg-Is-Field:  Field(?msg-field, ?pr)
    Rec-Is-Field:  Field(?rec-field, ?pr)
  indexes
    Write-Record when Msg-Is-Field, Msg-N-Dump

FILL-WITH-SPACES(Dest: ?pr)
  specializes MOVE(Source: 'Spaces, Dest: ?pr)

READ-ALL-RECORDS(File: ?f)
  components
    Reader:        READ-AND-RECORD-EOF(File: ?f, Ind: ?ind, Eof-Val: ?v)
    Looper:        LOOP-UNTIL-EQ-PLAN(Actions: ?seq, Flag: ?ind, Val: ?v)
  constraints
    Within-Loop:   ControlDep(?seq, Reader)
  indexes
    Reader when Within-Loop

  implies
    VALIDATE-INPUT-RECORDS(File: ?f, Cond: ?c)
      when
        Notifier:  NOTIFY-BAD-RECORD(Cond: ?c, Record: ?r, Msg: ?r)
      with
        In-Loop:   ControlDep(?seq, Notifier)
        Each-Read: ControlDep(Reader, Notifier)
        Same-File: RecordFile(?f, ?r)

  implies
    FILTER-COPY-INPUT(File: ?f, Cond: ?c)
      when
        Cond-Dump: COND-WRITE(Cond: ?c, Record: ?r)
      with
        In-Loop:   ControlDep(?seq, Cond-Dump)
        Each-Copy: ControlDep(Reader, Cond-Dump)
        Same-File: RecordFile(?f, ?r)
```

Figure 4. An example of several code patterns from our APU's plan library

process should always evaluate any constraint that will fail as soon as possible, since a single failed constraint eliminates a plan instance from further consideration, whereas a plan is recognized only when all constraints must succeed. Because indexing places a partial ordering on both matching (with the indexed component of the plan bound first) and constraint evaluation (with the indexing constraints evaluated first), the better the indexing constraints are as a predictor of a plan's presence, the fewer unneeded constraints will have to be evaluated. With DISPLAY-LABELLED-RECORD and our example program, the indexing happens to be 100 per cent effective: every successfully indexed instance of the plan turns out to be present in the program and has its remaining constraints succeed.

The performance improvement gained by indexing plans depends on a variety of factors, including the percentage of a plan library relevant to a given program, the effectiveness of the indexing constraints, the relative numbers of different program components, and the size of the program being understood. As of yet, we have not measured indexing's effectiveness in a way that takes all of these relevant factors into account. However, we have explored its performance on COBOL programs similar to our example (with up to 1,000 syntax-tree entries), running a Common-Lisp implementation of the understanding algorithm, using a plan library containing approximately 50 plans. On these programs, indexing the plan library resulted in reducing the amount of matching by a factor of ten, the number of constraint evaluations by a factor of five, and the overall time spent recognizing plans by a factor of five with precomputed data-dependencies and a factor of two with dynamically computed data-dependencies. In addition, as the programs to which we apply the plan library grow in size, indexing's advantage appears to grow steadily (although we must still explore whether this effect remains as the plan libraries increase in size and as we apply it to a wide variety of different sized programs). Regardless, however, these initial results suggest that our indexed program understanding algorithm is applicable to understanding program modules in the 1 000-line range and is worth studying further to determine what the limits of its applicability are.

Specialization

In addition to being defined in terms of components, plans in DECODE's library can be defined as *specializations* of other plans; that is, as a set of constraints on an existing plan's attributes. For example, the plan FILL-WITH-SPACES is defined as a specialization of a MOVE whose Source is SPACES. These specializations correspond to plans that contain a single component (the plan being specialized), that are indexed by that component, and that have constraints on that component's attributes. In fact, at definition time, these specializations are automatically translated into standard plan definitions.

The idea behind specializations is to make it easy to define one common class of plans and to encourage the definition and use of specialized plans as components and indexes in other plans. This simplifies the definition of higher-level plans that contain specialized plans as components by reducing the number of constraints that must be specified. This ability is simply a convenience, however, and has no implications on performance.

Implication

DECODE also allows plans to be defined as being conditionally implied by other plans. Any time it recognizes a plan that conditionally implies another plan, DECODE checks whether these conditions hold (which involves checking for additional components and evaluating additional constraints). For example, the plan READ-ALL-RECORDS implies the existence of the plan VALIDATE-INPUT-RECORDS when there exists an additional NOTIFY-BAD-RECORD that conceptually follows the READ-AND-RECORD-EOF within the input-reading loop.

The idea behind implications is to take advantage of small differences between the implementations of related plans, so that one plan can be recognized as a slight modification or extension to another. Essentially, plan implementations are organized in a discrimination net, which allows DECODE to use indexing to retrieve general plans to try first and then to use small, additional incremental tests to recognize more specific plans.

There are two alternatives to this approach. One is to have related plans be complete, stand-alone implementations that individually contain all necessary components and constraints. VALIDATE-INPUT-RECORDS and FILTER-COPY-INPUT, for example, could be defined so that each contains all of READ-ALL-RECORDS's components and constraints. This approach, however, leads to duplicate component matching and constraint evaluation that can be eliminated by explicit implication links. The other alternative is to have the specific plans contain the general plans as components and to have additional constraints that relate them to their other components. VALIDATE-INPUT-RECORDS and FILTER-COPY-INPUT, for example, could be defined to contain READ-ALL-RECORDS as one of their components. The problem with this approach is that the additional constraints may require access to READ-ALL-RECORDS's implementation (such as a control flow relationship involving its Reader or Looper), which then requires that READ-ALL-RECORDS have additional implementation-orientated attributes. Although this is just as efficient as implication links, it makes the definitions of plans much more difficult.

Aiding knowledge-based formation

DECODE also addresses the problem of forming a knowledge base that programmers can extend. It does so by extending the definition of a plan in two ways. First, it explicitly indicates whether each plan is *design-orientated* (of interest to the user) or *incremental* (necessary for the recognizer but not of general interest to the user). This is done by labelling each high-level plan definition with the operations it implements. OPEN-VALIDATE-CLOSE's definition, for example, indicates that it implements the conceptual operation ValidateRecords on the class RecordFile.

```
plan OPEN-VALIDATE-CLOSE(File Cond)
  isa         COMPLETE-FILE-PROCESSING-PLAN
  comments    "The plan of opening a file, reading through it,"
              "writing a message for each record that fails a"
              "particular test, and then closing the file."
  implements  ValidateRecords on RecordFile
```

When the design extractor recognizes an instance of any plan, it automatically adds the corresponding design elements to the knowledge base. This allows the design extractor to automatically recognize some conceptual design elements.

The design extractor is not always successful at recognizing high-level plans and instead may only be able to recognize a set of incremental plans. Even though these plans do not directly connect to abstract design elements, they can still aid the user in understanding the program. DECODE supports this by representing and displaying plans in the same way it represents and displays design elements. The user can then use the design editor to link these plans to user-provided design elements. DECODE also provides an explicit natural language description of each plan's function (which is simply a set of comments stored with the plan when it is defined). The user can then visually access this textual plan description as well as see what lines of the source file are relevant to the plan. This helps the user understand what abstraction the recognized plans capture.

4. THE STRUCTURED NOTEBOOK

Our design extractor can only recognize instances of plans that are contained within its plan library. Since the plan library is always incomplete, it is impossible for the design extractor to extract a complete design. As a result, DECODE provides a structured notebook for the user to use to continue the understanding process after DECODE's initial stab at extracting design information. The user's view of this structured notebook consists of two main windows: a code-browser, which displays the source code, and a design-editor, which displays a graphical view of the design. DECODE uses a simple automated graph layout algorithm to display the initial design derived by the automated program understander.

The design editor

Programmers record their understanding by interactively adding new conceptual design primitives and linking them to the program. The design-editor lets the user graphically perform the following object-orientated design actions:

- Create a class (e.g., ReservationRecord and TransactionFile) or operation (e.g., FilterCopy and Validate).
- Indicate that one class or operation is a subtype of another (e.g., the class ReservationFile *is a* File or ObtainNextTransaction *is a* ReadRecord operation).
- Indicate that a class is composed of subclasses (e.g. the class ReservationFile *contains* ReservationRecord.
- Indicate that a class has a particular operation (e.g., the class ReservationFile *has-op* ConstructionFromTransactionFile).
- Indicate that one operation makes use of another (e.g., the operation Validate for ReservationFile *uses-op* Validate for ReservationRecord).
- Indicate that an operation takes one or more instances of a class as input (e.g. ConstructFromTransactionFile *consumes* TransactionFile).
- Indicate that an operation generates one or more instances of a class as output (e.g. ConstructFromTransactionFile *produces* ErrorFile).
- Indicate that an operation has one or more implementations in the code (e.g., ValidateRecords for RecordFiles *has-impl* a particular set of statements).
- Indicate that a class has one or more object instantiations in the code (e.g., the class ReservationFile *has-inst* reservations-file).

These design actions capture many of the common relationships between objects recorded in typical graphical languages for recording object-orientated designs as they are created. All but the two actions that link classes and operations to their instances and implementations respectively can be done entirely independently of the source code being examined (and, in fact, are a subset of those found in some forward-engineering CASE tools). The linkage actions, however, are not generally supported by any of these tools. At best, the user is allowed to indicate that certain sections of code correspond to certain functionality in block diagrams of the system.

Linking source to design

The structured notebook allows a design element to be linked to either an arbitrary collection of statements or one or more plans.

A user links a design element to code by highlighting arbitrary (and possibly disjoint) sections of the program's source code in the code-browser window, pointing to an existing object-instance or implementation node, and finally invoking a 'Link' menu item. The result is that DECODE records that this design element is implemented by the highlighted source code lines.

To illustrate the linking process, suppose that from examining both the code and the automatically extracted plans, the user has recognized a new conceptual class ReservationsFile (a particular type of RecordFile) and a new conceptual operation, ConstructFromTransaction (the operation of building a ReservationsFile by running through a TransactionFile). Figure 5 shows how the user adds this design information. The user first uses the design-editor to add the new design elements (the classes ReservationsFile and TransactionFile and the operation ConstructFromTransaction) and then links them to highlighted chunks of source code in the browser window or to already recognized plans.

A user links a design element to a plan by creating a *uses-plan* relationship between them. This is useful when DECODE has automatically recognized a plan but is unable to automatically recognize the higher-level design elements that make use of it. This situation occurs when DECODE recognizes lower-level plans but not the higher-level plans that have them as components. In those cases where DECODE does recognize the higher-level plans that are related to design elements, it fills in the *uses-plan* relationships automatically.

The ability to link to existing plans lessens the need for users to identify individually all

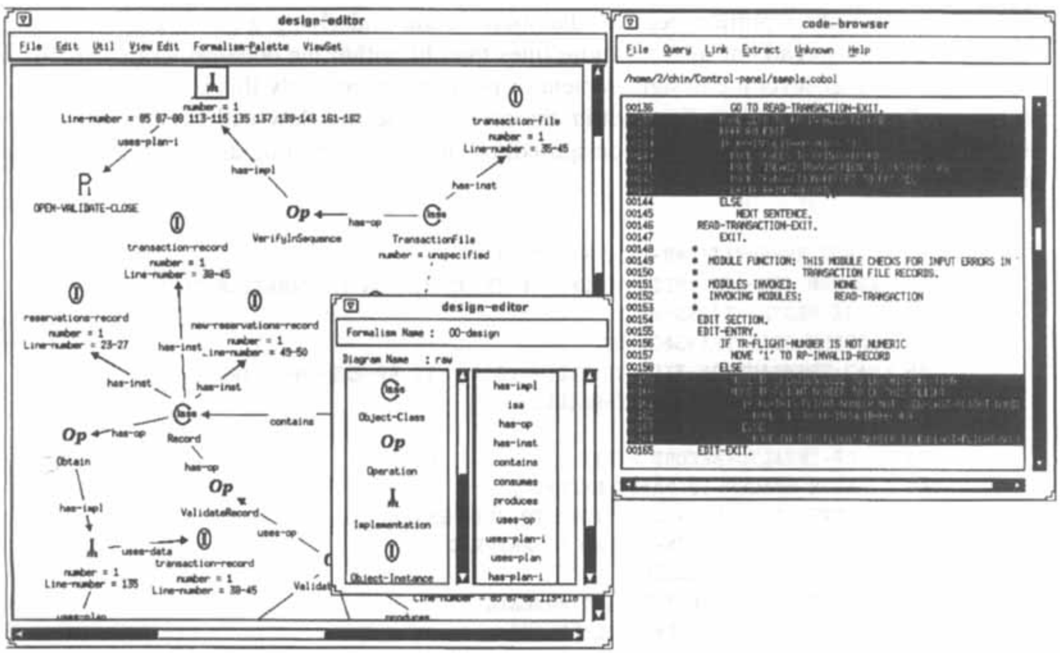

Figure 5. A screen dump showing how the user extends the initial system-extracted design

the statements related to a given design element. DECODE's automatic recognition of a portion of the planning structure underlying the program's design also eases the user's task, as it highlights the underlying relationships between possibly widely dispersed program statements, allowing the user to see the non-obvious connections between them.

In general, the ability to link design elements to source code or plans allows the user to record the underlying relationship between superficially unrelated statements. This ability is important, because just as the code that makes up a plan instance can be delocalized, so can the code that makes up a particular design element. For example, Figure 6 contains the pieces of the program source corresponding to an implementation of the conceptual operation ValidateRecords. These pieces are not contiguous, but instead are threaded through several widely separated paragraphs. By linking them to a single conceptual design element, the user essentially records a 'conceptual slice' of the program that captures the key portions of the program involved with validating input records. Having this information recorded is useful to any maintenance programmer assigned the task of adding additional input validations or verifying that particular validations are taking place.

Visualizing links between source and design

The design notebook currently indicates links between the design space and code in two ways.

One is that the code-browser window always displays a portion of the source code that is linked to the currently selected design element (where that link is either through *uses-plan* relationships to plans or directly to code). That is, to see what source code is linked to a particular design element, all the user has to do is select that design element in the design-editor window. This causes the corresponding lines in the code-browser window to be automatically highlighted (scrolling as needed to make the first corresponding line visible). In general, this is only a portion because the source code underlying a given plan (or design element) may be spread out over far more lines than fit within the code-browser window (in practice, the higher-level the design element or plan, the more likely this is to be a problem). Similarly, the user's selecting lines within the code-browser window will automatically highlight the corresponding nodes in the design-editor (likewise, scrolling as needed to make the node visible).

```
 86 OUTPUT RESERVATIONS-FILE PRINT-FILE
 87 PERFORM CRE-ATE UNTIL RP-END-OF-TRANS='1' OR TR-CREATION-CODE = '1'
 88 CLOSE RESERVATIONS-FILE
115 PERFORM READ-TRANSACTION
135 READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
137 MOVE ZERO TO RP-INVALID-RECORD
138 PERFORM EDIT
139 IF RP-INVALID-RECORD = '1'
140    MOVE SPACES TO PRINT-RECORD
141    MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
142    MOVE TRANSACTION-RECORD TO PRT-REC
143    WRITE PRINT-RECORD
156 IF TR-FLIGHT-NUMBER IS NOT NUMERIC
157    MOVE '1' TO RP-INVALID-RECORD
```

Figure 6. The underlying COBOL code making up the conceptual operation ValidateRecord

The other is through an explicit line-number attribute on each design element that lists the source lines to which it is linked. This attribute is filled-in automatically whenever a design element is linked to either plans or code. This method, however, is not entirely satisfactory for high-level plans which may involve many lines of source code. As a result, we are exploring the use of a window that provides a global visual representation of the source code and the pieces of it that are related to the current design element (others have used a similar technique to display program slices (Ball and Eick, 1994)).

Support for user queries

The result of the combined user/system understanding process is an extracted design and a set of links from this design to both the code and its underlying plan structure. In essence, this extracted design provides a queryable form of program documentation for use by maintenance programmers.

DECODE's notebook supports several different types of *conceptual* queries about the extracted design. These include;

Code function: *what is the purpose of a particular piece of source code?* (e.g., which conceptual operations require the data item TR-FLIGHT-NUMBER? Or, which conceptual operations involve a particular MOVE statement?)

Code location: *where is the code corresponding to a particular design element?* (e.g., what pieces of the source code implement the ReadRecord operation? What pieces of the source code are related to implementing any of the class TransactionRecord's operations?) These queries can be thought of as requesting *conceptual slices*, an alternative to structural program slicing (Weiser, 1982; Weiser, 1984).

Design completeness: *what design elements have not been recognized in the program?* (e.g., which ReservationFile operations are not implemented in this source file? Which lines of the program have not been linked to the design?)

One way to ask queries is simply to browse an existing design or source code. Selecting a line while browsing the source code corresponds to one type of code function query: 'what design element(s) capture this line of code?'. Similarly, selecting a design element while browsing the design corresponds to one type of code location query: 'which statement(s) implement this design element?'.

In addition to browsing, users can select queries from the 'Query' menu. In contrast to browsing, which is useful for finding out about a particular design element or a particular source code line, queries are useful for finding out information about a set of related design elements or about the source code or design as a whole. For example, when the user selects a particular class and invokes the 'Query' menu item, the system displays a window containing all of the source code involved in that class (recognized instances, as well as all the recognized implementations of the operations associated with that class).

Before carrying out any query, DECODE provides a selection box that allows the user to specify whether the user is interested in just this class or in all its subclasses. For example, specifying interest in just the File class results in nothing of interest for our example program, but selecting the File class and all its subclasses results in much of the program's data division being highlighted. This lets the user quickly determine which parts of the program are involved with different high-level conceptual classes.

The user is also given the option to see the inversion of any queries result, namely all code that is not involved in the selected class or operation. This allows the user to determine which parts of the program are not relevant to a given class or operation. Along the same lines, the user can use the code-browser's 'Unknown' menu to display all lines that have not been linked to any part of the object-orientated design. Likewise, the user can use this menu to ask DECODE to highlight any unconnected design elements. These queries help guide the user's attention to those parts of the source code that are not yet understood and to those parts of the design that have not been found implemented in the source code.

In addition to displaying the results of queries by highlighting source code lines and design elements, DECODE also provides users with an 'Extract' menu they can use to copy any highlighted lines into a separate window. This allows users to easily peruse lines that might be collected from a variety of locations in the original source code. The user can also save the extracted code into a file, which facilitates using existing code as the beginnings of a reuse library or an object-orientated reimplementation.

An example of conceptual querying

One final use of queries is to create a 'report' on a design element that includes both its source and all the design relationships it is involved in. Figure 7 is an example, where the user has requested a complete description of the ValidateRecords operation.

This query can be effortlessly asked by highlighting the ValidateRecords operation in the design window and then selecting the 'Query' menu item. The result is a window describing both conceptual design information (such as what types of objects this operation produces or consumes) as well as the particular source code statements that have been used to

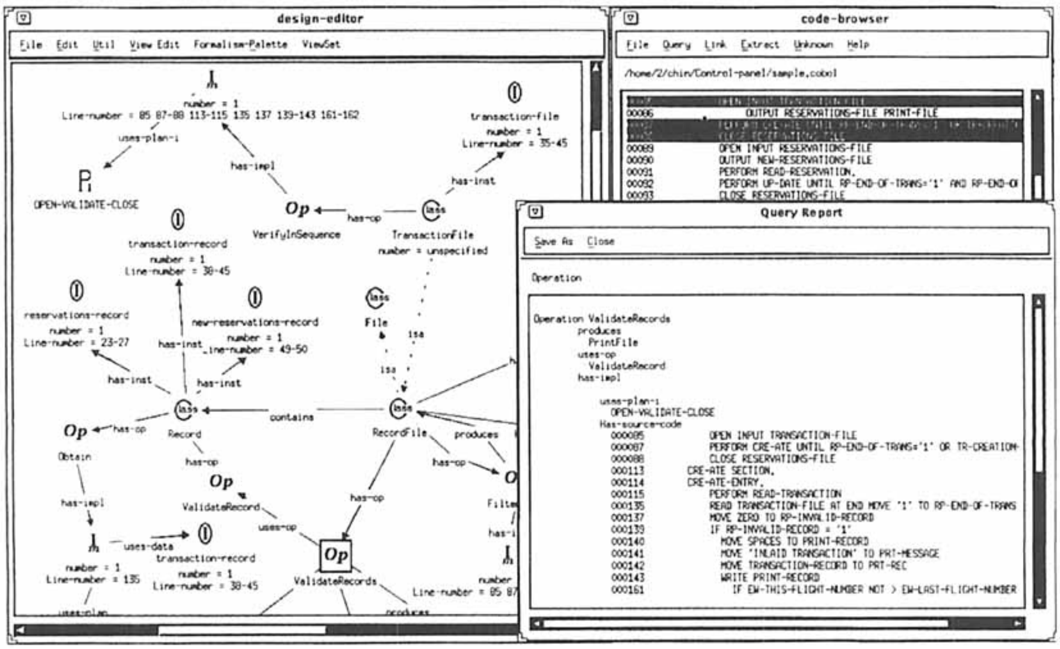

Figure 7. An example of the user requesting a complete description of the ValidateRecords operation

implement it. In addition, the relevant source code lines are highlighted in the code browser. By browsing and querying the design, the user can quickly come to understand where various high-level functionality is actually implemented within the source code.

Maintenance uses of conceptual queries

Obtaining answers to these different types of queries supports various maintenance tasks.

Code function queries support impact analysis. While existing data flow or control flow tools can highlight the statements affected by a proposed modification, they provide no information about the underlying purpose of the affected statements. By asking a code function query, however, the programmer can understand exactly which other design elements the change impacts.

Similarly, code location queries support maintenance programmers in determining where to make a particular modification. They allow the search for relevant existing code to take place in terms of high-level design elements rather than through a laborious low-level search through the code.

Finally, design completeness queries support maintenance programmers in locating the places where an existing program differs from a stereotypical design for its general class of application. However, they are also useful while forming the knowledge base describing a legacy program's design, as they can be used to determine how well the program has been understood and to find the places where further understanding may be necessary.

DECODE can provide answers to these queries only because it explicitly represents the links between the conceptual design and the actual implementation and provides the user with a mechanism for recording these links. Given these links, however, all these queries can be answered quite efficiently, essentially with simple graph-traversal techniques. For example, displaying all code relevant to a particular class is simply a matter of traversing all its subclasses in the design graph, and then highlighting all lines associated with each of its operations.

Reconfiguration by Meta editor

We have chosen a particular object-orientated framework for representing our designs. DECODE, however, is not dependent on this framework. Instead, users can easily specify their own design frameworks.

To support 'design independence', DECODE's graphical design-editor is an implementation of the MERA (Meta Entity Relation Attribute) language (Takeda, Chin and Miyamoto, 1992). In particular, the design-editor implements the Meta language protocol, which allows users to easily reconfigure the design-editor by using it to edit a Meta diagram that describes the desired configuration. Figure 8 shows the Meta diagram used to define the configuration of DECODE's design-editor. The Meta diagram describes the types of entities (represented by an 'E' icon), attributes (represented by an 'A' icon), and relations (represented by arcs). Which entities or relations have which attributes are defined by Has_Attribute arcs between the entity/relation and the attribute. In Figure 8, the entities include Object-Class, Operation, and Implementation, among others. Relations include has-impl, isa, and has-op, among others. The entities Object-Instance and Implementation are defined to have the attribute, Line-number.

Using the Meta editor capabilities of the design-editor, end users can easily reconfigure the design-editor to encode their own design formats. By using multiple Meta diagrams, users

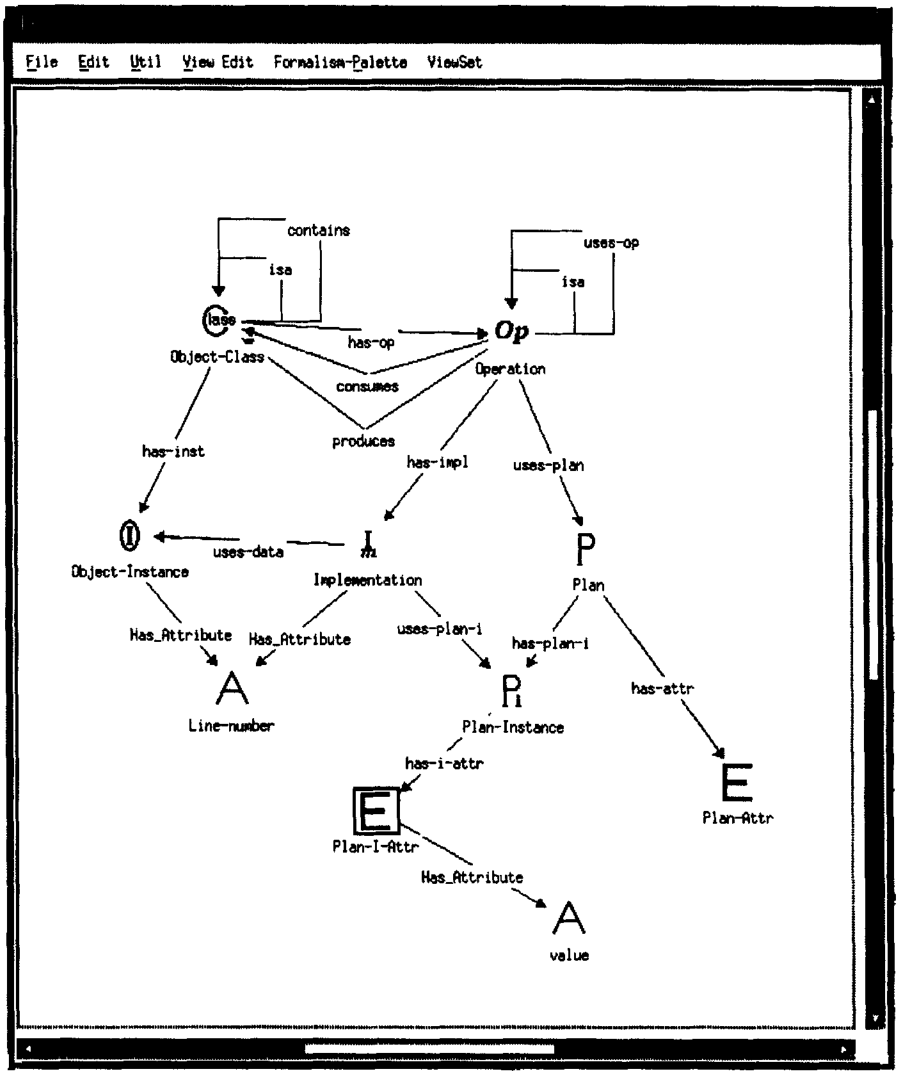

Figure 8. The Meta diagram for DECODE's design-editor

can even create multiple design-editors to edit multiple types of diagrams, each describing different design aspects of the legacy software. Most importantly, however, it means that our co-operative understanding environment is not tuned to the particular object-orientated design framework we have initially chosen.

5. USING DECODE TO EXTRACT DESIGNS

We have observed two distinctly different paradigms for using the design-editor to record design information.

One is top-down: the user attempts to map an initial design for the program to the code. This initial design may be one hypothesized by the user based on a combination of past

domain experience and some existing external information about what the program does. With our example COBOL program, this approach was taken by those users who were familiar with batch-orientated master file updates. They assumed there would be code for reading through a transaction file, validating records, and then copying certain records through the master file. They then went looking for the code that performed these tasks. (This approach was pointed out in Biggerstaff, Mitbander and Webster (1994), along with a mechanism for automatically detecting the presence of such high-level design elements in the code). DECODE supports this approach by letting users first record these initial design assumptions by creating appropriate design elements and by then letting users link these design elements to the code as their implementations are located. However, DECODE lets the user modify the design at any time during the understanding process, as is necessary when unanticipated design elements are recognized.

The user's hypothesized initial design may also be a very standard design, one that is either domain-independent or shared by many programs within a particular domain, such as a typical linked-list object or a standard approach to updating a master file from a transaction file. In this case, the entire standard design can be recorded in advance and loaded by different users attempting to match the design to existing code.

The other approach to program understanding is much more bottom-up: the user slowly works through the code, recording classes and operations only as they are discovered. This approach is most appropriate when there is little knowledge in advance about how the program might be designed. It is also the approach taken by most students given the task of understanding our example program, as they usually had no previous experience with transaction processing problems. DECODE supports this approach by performing its own initial bottom-up understanding of the code and by allowing the user to create design elements whenever the user happens to notice one's appearance in the code.

Both of these paradigms lead to constantly revised extracted designs as the program becomes better understood. As a result, our design-editor allows users to delete, rename, and otherwise modify the design on the fly. This is especially important as a user may recognize an object or operation in the code and then later realize that this recognition was a mistake and that some other object or operation is actually there. With this ability to constantly update and revise the underlying design, at any point in time the extracted design simply represents a best guess as to what portions of the software do, and errors can easily be corrected.

One drawback with DECODE is that either approach to design extraction can very quickly lead to a large collection of different design elements, even for relatively small programs. This problem leads to the users having to spend considerable time perusing the design each time they wish to add a new design element (to ensure it is not already there) or want to link code to an existing design element (to find an element they know is there). To address this scalability problem we need to augment DECODE with mechanisms for specifying which design elements should be visible in the design-editor window at any point in time, as well as providing tools, such as key word search mechanisms, to aid in locating desired design elements.

6. RELATED WORK

Our work is most closely related to two systems, COBOL/SRE (Ning, Engberts and Kozaczyzski, 1993) and LaSSIE (Devanbu *et al.*, 1990).

COBOL/SRE

COBOL/SRE (Ning, Engberts and Kozacyzski, 1993) allowed its users to create *segments* by highlighting arbitrary sections of code or by requesting various types of code slices. These segments can then be combined using set-like insertion, union and difference operations, extracted from code into separate files, and stored along with comments describing their function.

A key extension in DECODE is that rather than having users provide English 'comments' that describe what segments do, DECODE forces them to associate segments with operations in a design hierarchy. This requires more work on the part of the user, but it allows DECODE to answer conceptual queries about segments. In contrast, COBOL/SRE optionally allows the user to associate each segment with a comment, but provides no mechanism for retrieving segments based on the contents of those comments. In a sense, DECODE can be thought of as augmenting COBOL/SRE with a visual design description language that the user can use to comment what a segment does, as well as query mechanisms that exploit this description to allow the user to retrieve segments with particular characteristics.

The other key extension in DECODE is its indexed, code-driven program understanding mechanism. DECODE's representation for programming plans directly extends COBOL/SRE's representation to make it suitable for code-driven rather than library-driven understanding. In addition, it extends COBOL/SRE's representation to automatically generate object-orientated descriptions of code segments, rather than having the understander stop at the level of program plans.

One relative weakness in DECODE is that COBOL/SRE's mechanisms for segment construction are considerably more powerful, especially the ability to create a segment corresponding to a program slice or from the call tree resulting from a single PERFORM and the ability to use set operations to combine arbitrary segments. It is, however, relatively straightforward to add these mechanisms to DECODE and doing so should make it easier for users to indicate which statements correspond to a given design element.

LaSSIE

LaSSIE (Devanbu *et al.*, 1990) uses a classification hierarchy to describe the tasks accomplished by the modules that make up a large phone-switching software system. It uses this hierarchy to answer queries from programmers trying to locate modules implementing particular conceptual functions.

LaSSIE provides a more pwoerful query-answering capability than DECODE's (since objects can be located by providing arbitrary constraints on types and attributes), but has the drawback that it requires a knowledge engineer to create the formal description of the software, an approach that relies on pre-existing documentation often unavailable in legacy systems. DECODE's design description language can be viewed as a compromise: it results in a less detailed description about what pieces of the program do, which means it supports much less-powerful queries, but it provides a visual mechanism that allows programmers to produce these descriptions rather than requiring knowledge engineers.

LaSSIE also differs from DECODE in that it is designed to describe programs at the module level, not at the statement level. As a result, programmers can only ask detailed queries about module components, not about arbitrary code fragments. These types of detailed questions, however, are relevant to programmers trying to understand legacy software systems because answering them is essential for modification, reimplementation or porting.

One final difference is that LaSSIE made no attempt to extract conceptual information automatically. In contrast, DECODE explicitly tries to integrate automated program understanding techniques with assisted ones, and it makes the design information it does extract available to the user forming the design description.

7. FUTURE WORK

DECODE represents a potential advance in tools for dealing with legacy software, primarily in its integration of automated and assisted program recognition and in its careful attention to scaling issues in automated program understanding. However, DECODE does not yet address several important issues essential for re-engineering and maintaining complex legacy software systems.

Support for extraction of complete object-oriented designs

The only design elements DECODE now extracts and records involve classes and operations. However, a more complete object-orientated design also includes dynamic models, such as explicit state transition diagrams that show the order in which operations occur, the conditions under which they occur, and so on (Rumbaugh *et al.*, 1991). These high-level state diagrams represent the program's control flow at the conceptual level rather than at the statement level.

DECODE needs to be augmented to recognize and record these object-orientated state transition diagrams. This will involve extending our design-editor to allow users to construct the state diagrams and link them to recognized design elements. This will also involve modifying DECODE so that as users traverse these state diagrams, it will automatically highlight the corresponding program source code. Doing so will allow the user to see the relationship between conceptual states and particular statements in the legacy system. For example, the user will be able to obtain answers to questions such as 'what happens before (or after) the FilterCopy operation is applied to transaction-file?'.

This will also require that we extend our automatic program understanding component to recognize and record conceptual control-flow information (i.e., states and state transitions). Our current plan is to use the abstract control-flow information we maintain about recognized plans to determine the flow relationships between the operations these plans implement.

Support for programmer-specified code patterns

DECODE currently concentrates on the recognition of domain-independent code patterns and assumes that all these code patterns will be provided by knowledge engineers. As a result, there is no provision to allow non-expert users to easily add new code patterns to the library. However, in large complex legacy systems, large sections of code could be recognized using domain-dependent code patterns, which means that much larger pattern libraries are needed. Because of the large variety of possible domains, it would be impossible to preconstruct such code pattern libraries. Instead, the process of adding new code patterns must be simplified and supported by knowledge acquisition tools.

We are currently exploring letting users provide code patterns by example. The idea is to have users select a program section that corresponds to an instance of a particular design element. This code section can be considered an overly constrained code pattern (that will

only recognize this one instance). The system can then provide the programmer with a list of all the specific constraints involved in the example, which the programmer can edit to create a final code pattern. While we have not included this mechanism in our current system, it suggests a path by which a sizeable code pattern library can be formed without using specialized knowledge engineers.

Providing additional co-operation

DECODE's co-operation in the design extraction process is now somewhat limited. Its currently 'co-operates' by initially extracting as much design information as it can automatically and by providing an easy to use visual environment for recording and locating extracted design information.

There are, however, a variety of specific places where we can extend DECODE to be more co-operative. For example, when the user highlights a PERFORM as part of a recognized design element. DECODE should automatically highlight the PERFORMed statements and include them as part of the design element's implementation. Similarly, when the user extracts the portion of the source code that implements a particular design element, DECODE should also automatically extract the relevant data declarations. These are both straightforward extensions given existing tools for extracting data and control flow information.

We are also exploring some general ways DECODE could be made more co-operative. DECODE's understander now essentially recognizes a design element only when it can prove that it is present, and it does this understanding just once, before the user starts examining the program. But a reasonable alternative is to have DECODE try to relate user-suggested design elements to the code but with relaxed constraints and by allowing missing components. This can provide a mechanism for the user to test hypotheses that certain elements are present, while providing DECODE with a tractable search space to examine.

Providing additional support for maintenance

DECODE currently assumes the program being understood will not change—an assumption that clearly does not hold with real-world programs. As a result, DECODE currently has no mechanism for recognizing which parts of the program (and their corresponding design elements) are affected by a given change. The problem is that this makes DECODE unusable as a real-world maintenance tool (although it does not detract from its usefulness as a tool for exploring co-operative program understanding).

Fortunately, this problem is relatively easy to address. We need to modify DECODE so that whenever a program is edited, DECODE determines which lines were changed, which previously recognized plans are no longer present, and which design elements were linked to those lines. DECODE can then indicate those lines and design elements to the user, and the user can then determine whether the modified code is relevant to those design elements or whether the modification reflects a change in design that must be recorded in the design-editor. In addition, DECODE must also rerun its plan recognition engine to try to recognize any new plans that are now relevant based on the modified source code.

8. CONCLUSIONS

A completely automated, pattern-based program understanding system would be an ideal aid to maintenance programmers. Unfortunately, that ideal is likely to be impossible to attain.

As a result, it is crucial to begin blending automated and assisted program understanding mechanisms. This paper has presented DECODE, a prototype environment intended to support programmers who are understanding and extracting designs from large, real-world legacy systems. DECODE includes an automatic program recognition component that explicitly addresses scaling issues, a structured notebook for recording system and programmer observations about a program's design and its relationship to code, and a simple but powerful menu-based query mechanism.

DECODE is only a prototype (implemented in a combination of Common Lisp, C, C++ and Motif), its current code patterns are targeted to automatic extraction of domain-independent design information, and it supports recording only a limited amount of design information. Despite these limitations, however, it provides benefits to the users we have had use it to understand relatively small programs. The code patterns easily detect common constructs, such as input-validation loops, despite their being distributed over many different lines of code. Even this minimal amount of automated understanding eases the user's task in figuring out what a program does and how it works. In addition, DECODE's simple capabilities for visually recording user-extracted design information and linking it to code allows it to answer a wide range of *conceptual* queries, and this conceptual knowledge is crucial to being able to successfully modify, extend, or translate a legacy system.

The key idea in DECODE, however, is that the system and programmer work together to understand the software. Automated program understanders are now capable of recognizing many common low-level design elements in the code. Programmers, on the other hand, are quite capable of imposing a high-level design framework on a piece of software and trying to see where the elements of that framework are implemented. DECODE tries to combine both capabilities to speed the process of completely understanding a piece of legacy software. Our hope is that this co-operative approach will significantly reduce the time and effort it takes for programmers to perform the program understanding tasks that are a prerequisite to maintaining legacy systems.

Acknowledgements

This project is supported by the KBSA project, Air Force Rome Labs, under Air Force contract #F30602-93-C-0257. We gratefully acknowledge W. Kozacyznski, J. Ning and A. Engberts of Andersen Consulting for providing us with COBOL/SRE upon which much of our work is based. Jianqun Cheng and Jeremy T. Harrison were responsible for implementation of significant portions of this project.

APPENDIX: PART OF A COBOL PROGRAM TO UNDERSTAND

```
000000 IDENTIFICATION DIVISION.
        PROGRAM-ID.  RESVPG.
        AUTHOR.  MIYAMOTO.
        *
        *  THIS PROGRAM IS THE IMPLEMENTATION FOR THE RESERVATIONS
        *  PROGRAM.
        *
         ENVIRONMENT DIVISION.
         CONFIGURATION SECTION.
         SOURCE-COMPUTER.
         OBJECT-COMPUTER.
         INPUT-OUTPUT SECTION.
         FILE-CONTROL.
             SELECT RESERVATIONS-FILE     ASSIGN TO MASS-STORAGE REF.
             SELECT TRANSACTION-FILE      ASSIGN TO MASS-STORAGE TR.
             SELECT PRINT-FILE            ASSIGN TO PRINTER.
             SELECT NEW-RESERVATIONS-FILE ASSIGN TO MASS-STORAGE NR.
         DATA DIVISION.
         FILE SECTION.
         FD  RESERVATIONS-FILE
             LABEL RECORDS ARE STANDARD
             DATA RECORD IS RESERVATIONS-RECORD.
         01  RESERVATIONS-RECORD.
           05  RR-REC.
             10  RR-FLIGHT-NUMBER       PIC 9(6).
             10  RR-PASSENGER-NAME1     PIC X(30).
             10  RR-PASSENGER-NAME2     PIC X(30).
         FD  PRINT-FILE
             LABEL RECORDS ARE  OMITTED
             DATA RECORD IS PRINT-RECORD.
         01  PRINT-RECORD.
           05  PRT-MESSAGE              PIC X(30).
           05  PRT-REC                  PIC X(68).
           05  FILLER                   PIC X(34).
         FD  TRANSACTION-FILE
             LABEL RECORDS ARE OMITTED
             DATA RECORD IS TRANSACTION-RECORD.
         01  TRANSACTION-RECORD.
           05  TR-REC.
             10  TR-FLIGHT-NUMBER       PIC 9(6).
             10  TR-PASSENGER-NAME1     PIC X(30).
             10  TR-PASSENGER-NAME2     PIC X(30).
           05  TR-CREATION-CODE         PIC X.
           05  TR-TRANSACTION-CODE      PIC X.
           05  FILLER                   PIC X(12).
         FD  NEW-RESERVATIONS-FILE
             LABEL RECORDS ARE STANDARD
             DATA RECORD IS NEW-RESERVATIONS-RECORD.
         01  NEW-RESERVATIONS-RECORD.
           05  NRR-REC                  PIC X(66).
         WORKING-STORAGE SECTION.
         01  RESERVATIONS-PROGRAM-WORK.
           05  RP-END-OF-TRANS          PIC X VALUE ZERO.
           05  RP-END-OF-RESERV         PIC X VALUE ZERO.
           05  RP-INVALID-RECORD        PIC X VALUE ZERO.
         01  EDIT-WORK.
           05  EW-THIS-FLIGHT-NUM.
             10  EW-THIS-CREATION       PIC X.
             10  EW-THIS-FLIGHT         PIC 9(6).
           05  EW-THIS-FN REDEFINES EW-THIS-FLIGHT-NUM.
```

```
              10  EW-THIS-FLIGHT-NUMBER  PIC 9(7).
            05  EW-FLIGHT-NUMBER         VALUE ZERO.
              10  EW-CREATION            PIC X.
              10  EW-FLIGHT              PIC 9(6).
            05  EW-FN REDEFINES EW-FLIGHT-NUMBER.
              10  EW-LAST-FLIGHT-NUMBER       PIC 9(7).
      /
       PROCEDURE DIVISION.
      *
      *  MODULE FUNCTIONS: THIS IS THE MAINLINE MODULE OF THE
      *                    RESERVATIONS PROGRAM.  IT INVOKES MODULES TO
      *                    INITIALLY CREATE A RESERVATIONS FILE, TO
      *                    UPDATE THIS FILE WITH NEW RESERVATIONS AND
      *                    CANCELLATIONS, AND TO PRINT OUT THE CONTENTS
      *                    THE NEW UPDATED RESERVATIONS FILE.
      *  CONTROL VARIABLES MODIFIED: RP-END-OF-RESERV
      *  MODULES INVOKED:            CRE-ATE
      *                              READ-RESERVATION
      *                              UP-DATE
      *                              RE-PORT
      *  INVOKING MODULES:           NONE
      *
       RESERVATIONS-PROGRAM SECTION.
       RESERVATIONS-PROGRAM-ENTRY.
           OPEN INPUT TRANSACTION-FILE
                OUTPUT RESERVATIONS-FILE PRINT-FILE
           PERFORM CRE-ATE UNTIL RP-END-OF-TRANS='1' OR TR-CREATION-CODE = '1'.
           CLOSE RESERVATIONS-FILE
           OPEN INPUT RESERVATIONS-FILE
           OUTPUT NEW-RESERVATIONS-FILE
           PERFORM READ-RESERVATION.
           PERFORM UP-DATE UNTIL RP-END-OF-TRANS='1' AND RP-END-OF-RESERV = '1'.
           CLOSE RESERVATIONS-FILE
                 NEW-RESERVATIONS-FILE
                 TRANSACTION-FILE
           OPEN INPUT NEW-RESERVATIONS-FILE
           MOVE ZERO TO RP-END-OF-RESERV
           MOVE SPACES TO PRINT-RECORD
           MOVE 'UPDATED RESERVATION' TO PRT-MESSAGE
           PERFORM RE-PORT UNTIL RP-END-OF-RESERV = '1'.
           CLOSE NEW-RESERVATIONS-FILE PRINT-FILE
           STOP RUN.
       RESERVATIONS-PROGRAM-EXIT.
           EXIT.

      *
      *  MODULE FUNCTION: THIS MODULE INITIALLY CREATES THE RESERVATIONS
      *                   FILE FORM TRANSACTION RECORDS.
      *  CONTROL VARIABLES MODIFIED: NONE
      *  MODULES INVOKED:            READ-TRANSACTION
      *  INVOKING MODULES:           RESERVATIONS-PROGRAM
      *
       CRE-ATE SECTION.
       CRE-ATE-ENTRY.
           PERFORM READ-TRANSACTION
           IF RP-END-OF-TRANS = ZERO AND RP-INVALID-RECORD = ZERO AND
              TR-CREATION-CODE NOT = '1'
             WRITE RESERVATIONS-RECORD FROM TR-REC
           ELSE
             NEXT SENTENCE.
       CRE-ATE-EXIT.
           EXIT.
      *
      *  MODULE FUNCTION: THIS MODULE READS A RECORD FROM THE
      *                   TRANSACTION FILE. AT FILE END,
```

```
*                   RP-END-OF-TRANS IS SET TO 1.
*  CONTROL VARIABLES MODIFIED: RP-END-OF-TRANS
*  MODULES INVOKED:            EDIT
*  INVOKING MODULES:           CRE-ATE
*                              FINISH-TRANSACTION
*                              MATCH-FLIGHT
*
 READ-TRANSACTION SECTION.
 READ-TRANSACTION-ENTRY.
     READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
       GO TO READ-TRANSACTION-EXIT.
     MOVE ZERO TO RP-INVALID-RECORD
     PERFORM EDIT
     IF RP-INVALID-RECORD = '1'
       MOVE SPACES TO PRINT-RECORD
       MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
       MOVE TRANSACTION-RECORD TO PRT-REC
       WRITE PRINT-RECORD
     ELSE
       NEXT SENTENCE.
 READ-TRANSACTION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE CHECKS FOR INPUT ERRORS IN THE
*                   TRANSACTION FILE RECORDS.
*  MODULES INVOKED:       NONE
*  INVOKING MODULES:      READ-TRANSACTION
*
 EDIT SECTION.
 EDIT-ENTRY.
     IF TR-FLIGHT-NUMBER IS NOT NUMERIC
       MOVE '1' TO RP-INVALID-RECORD
     ELSE
       MOVE TR-CREATION-CODE TO EW-THIS-CREATION
       MOVE TR-FLIGHT-NUMBER TO EW-THIS-FLIGHT
         IF EW-THIS-FLIGHT-NUMBER NOT > EW-LAST-FLIGHT-NUMBER
           MOVE '1' TO RP-INVALID-RECORD
         ELSE
           MOVE EW-THIS-FLIGHT-NUMBER TO EW-LAST-FLIGHT-NUMBER.
 EDIT-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE READS A RESERVATIONS FILE RECORD.
*                   AT FILE END, RP-END-OF-RESERV IS SET TO 1.
*  CONTROL VARIABLES MODIFIED: RP-END-OF-RESERV
*  MODULES INVOKED:            NONE
*  INVOKING MODULES:           RESERVATIONS-PROGRAM
*                              FINISH-RESERVATION
*                              MATCH-FLIGHT
*
 READ-RESERVATION SECTION.
 READ-RESERVATION-ENTRY.
     READ RESERVATIONS-FILE AT END MOVE '1' TO RP-END-OF-RESERV.
 READ-RESERVATION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE UPDATES THE RESERVATIONS FILE
*                   WITH NEW TRANSACTIONS TO CREATE A NEW UPDATED
*                   RESERVATION FILE.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            FINISH-RESERVATION
*                              FINISH-TRANSACTION
*                              MATCH-FLIGHT
*  INVOKING MODULES:           RESERVATIONS-PROGRAM
*
```

```
       UP-DATE SECTION.
       UP-DATE-ENTRY.
           IF RP-END-OF-TRANS = '1' AND RP-END-OF-RESERV = ZERO
             PERFORM FINISH-RESERVATION UNTIL RP-END-OF-RESERV = '1'
           ELSE IF RP-END-OF-TRANS = ZERO AND
             RP-END-OF-RESERV = '1'
             PERFORM FINISH-TRANSACTION UNTIL RP-END-OF-TRANS = '1'
           ELSE IF RP-END-OF-TRANS = ZERO AND
             RP-END-OF-RESERV = ZERO
             PERFORM MATCH-FLIGHT
           ELSE
             NEXT SENTENCE.
       UP-DATE-EXIT.
           EXIT.
      *
      *  MODULE FUNCTION: WHEN THERE ARE NO MORE TRANSACTIONS TO PROCESS
      *                   THIS PART OF THE UPDATE PROCESS WRITES OUT
      *                   THE REMAINING RESERVATIONS ON THE NEW UPDATED
      *                   RESERVATIONS FILE.
      *  CONTROL VARIABLES MODIFIED: NONE
      *  MODULES INVOKED:            WRITE-NEW-RESER
      *                              READ-RESERVATION
      *  INVOKING MODULES:           UP-DATE
      *
       FINISH-RESERVATION SECTION.
       FINISH-RESERVATION-ENTRY.
           PERFORM WRITE-NEW-RESER
           PERFORM READ-RESERVATION.
       FINISH-RESERVATION-EXIT.
           EXIT.
      *
      *  MODULE FUNCTION: THIS MODULE WRITES A RECORD OUT ON THE NEW
      *                   UPDATED RESERVATIONS FILE.
      *  CONTROL VARIABLES MODIFIED: NONE
      *  MODULES INVOKED:            NONE
      *  INVOKING MODULES:           FINISH-RESERVATION
      *                              MATCH-FLIGHT
      *                              UPDATE-RESERVATION
      *
       WRITE-NEW-RESER SECTION.
       WRITE-NEW-RESER-ENTRY.
           MOVE RR-REC TO NRR-REC
           WRITE NEW-RESERVATIONS-RECORD.
       WRITE-NEW-RESER-EXIT.
           EXIT.
      *
      *  MODULE FUNCTION: WHEN THERE ARE NO MORE RESERVATION RECORD TO
      *                   UPDATE.  THIS PART OF THE UPDATE PROCESS
      *                   WRITES OUT NEW RESERVATIONS RECORDS FROM THE
      *                   REMAINING TRANSACTIONS.
      *  CONTROL VARIABLES MODIFIED: NONE
      *  MODULES INVOKED:            WRITE-TRANS-TO-RESER
      *                              READ-TRANSACTION
      *  INVOKING MODULES:           UP-DATE
      *
       FINISH-TRANSACTION SECTION.
       FINISH-TRANSACTION-ENTRY.
           IF RP-INVALID-RECORD = ZERO
             IF TR-TRANSACTION-CODE = '1'
               PERFORM WRITE-TRANS-TO-RESER
               MOVE SPACES TO PRINT-RECORD
               MOVE 'RESERVATION ADDED' TO PRT-MESSAGE
               MOVE TRANSACTION-RECORD TO PRT-REC
               WRITE PRINT-RECORD
             ELSE
```

```
         NEXT SENTENCE
     ELSE
       NEXT SENTENCE.
     PERFORM READ-TRANSACTION.
 FINISH-TRANSACTION-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE WRITES A RECORD OUT ON THE NEW
*                   UPDATED RESERVATIONS FILE USING A
*                   TRANSACTION RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  INVOKING MODULES:           FINISH-TRANSACTION
*                              MATCH-FLIGHT
*  MODULES INVOKED:            NONE
*
 WRITE-TRANS-TO-RESER SECTION.
 WRITE-TRANS-TO-RESER-ENTRY.
     MOVE TR-REC TO NRR-REC
     WRITE NEW-RESERVATIONS-RECORD.
 WRITE-TRANS-TO-RESER-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE TRIES TO MATCH AN EXISTING
*                   RESERVATIONS FILE RECORD WITH A TRANSACTION
*                   FILE RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            WRITE-NEW-RESER
*                              READ-RESERVATION
*                              WRITE-TRANS-TO-RESER
*                              READ-TRANSACTION
*                              UPDATE-RESERVATION
*  INVOKING MODULES:           UP-DATE
*
 MATCH-FLIGHT SECTION.
 MATCH-FLIGHT-ENTRY.
     IF RP-INVALID-RECORD = '1'
       PERFORM READ-TRANSACTION
     ELSE IF RR-FLIGHT-NUMBER < TR-FLIGHT-NUMBER
       PERFORM WRITE-NEW-RESER
       PERFORM READ-RESERVATION
     ELSE IF RR-FLIGHT-NUMBER > TR-FLIGHT-NUMBER
       MOVE SPACES TO PRINT-RECORD
       MOVE 'RESERVATION RECORD ADDED' TO PRT-MESSAGE
       MOVE TRANSACTION-RECORD TO PRT-REC
       WRITE PRINT-RECORD
       PERFORM WRITE-TRANS-TO-RESER
       PERFORM READ-TRANSACTION
     ELSE IF RR-FLIGHT-NUMBER = TR-FLIGHT-NUMBER
       PERFORM UPDATE-RESERVATION
       PERFORM READ-RESERVATION
       PERFORM READ-TRANSACTION
     ELSE
       NEXT SENTENCE.
 MATCH-FLIGHT-EXIT.
     EXIT.
*
*  MODULE FUNCTION: THIS MODULE UPDATES A RESERVATIONS FILE
*                   RECORD.
*  CONTROL VARIABLES MODIFIED: NONE
*  MODULES INVOKED:            MOD-IFY
*                              WRITE-NEW-RESER
*  INVOKING MODULES:           MATCH-FLIGHT
*
 UPDATE-RESERVATION SECTION.
 UPDATE-RESERVATION-ENTRY.
```

```
           IF TR-TRANSACTION-CODE = ZERO
           PERFORM MOD-IFY
           IF RR-PASSENGER-NAME1 = SPACES AND
              RR-PASSENGER-NAME2 = SPACES
             MOVE SPACES TO PRINT-RECORD
             MOVE 'RESERVATION RECORD DELETED' TO PRT-MESSAGE
             MOVE RESERVATIONS-RECORD TO PRT-REC
             WRITE PRINT-RECORD
           ELSE
             PERFORM WRITE-NEW-RESER
           ELSE IF RR-PASSENGER-NAME1 = SPACES OR
                   RR-PASSENGER-NAME2 = SPACES
             PERFORM MOD-IFY
             MOVE SPACES TO PRINT-RECORD
             MOVE 'RESERVATION ADDED' TO PRT-MESSAGE
             MOVE TRANSACTION-RECORD TO PRT-REC
             WRITE PRINT-RECORD
           ELSE
             MOVE SPACES TO PRINT-RECORD
             MOVE 'NO ROOM ON FLIGHT' TO PRT-MESSAGE
             MOVE TRANSACTION-RECORD TO PRT-REC
             WRITE PRINT-RECORD.
       UPDATE-RESERVATION-EXIT.
              EXIT.
      *
      * MODULE FUNCTION: THIS MODULE CHANGES THE PASSENGER NAMES IN
      *                  THE RESERVATIONS FILE RECORD ACCORDING TO NEW
      *                  INFORMATION FROM A MATCHING TRANSACTION FILE
      *                  RECORD.
      * CONTROL VARIABLES MODIFIED: NONE
      * MODULES INVOKED:            NONE
      * INVOKING MODULES:           UPDATE-RESERVATION
      *
       MOD-IFY SECTION.
       MOD-IFY-ENTRY.
           IF TR-TRANSACTION-CODE = ZERO
             IF TR-PASSENGER-NAME1 = RR-PASSENGER-NAME1 OR
                TR-PASSENGER-NAME2 = RR-PASSENGER-NAME1
               MOVE SPACES TO RR-PASSENGER-NAME1
               GO TO MOD-IFY-EXIT
             ELSE IF TR-PASSENGER-NAME1 = RR-PASSENGER-NAME2 OR
                     TR-PASSENGER-NAME2 = RR-PASSENGER-NAME2
               MOVE SPACES TO RR-PASSENGER-NAME2
               GO TO MOD-IFY-EXIT
             ELSE
               GO TO MOD-IFY-EXIT
             ELSE
               NEXT SENTENCE.
             IF RR-PASSENGER-NAME1 = SPACES
               MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME1
             ELSE
               MOVE TR-PASSENGER-NAME1 TO RR-PASSENGER-NAME2
             IF TR-PASSENGER-NAME2 NOT = SPACES
               IF RR-PASSENGER-NAME2 = SPACES
                 MOVE TR-PASSENGER-NAME2 TO RR-PASSENGER-NAME2
           ELSE
             MOVE SPACES TO PRINT-RECORD
             MOVE 'NO ROOM FOR 2ND PASSENGER' TO PRT-MESSAGE
             MOVE TRANSACTION-RECORD TO PRT-REC
             WRITE PRINT-RECORD
           ELSE
             NEXT SENTENCE.
       MOD-IFY-EXIT.
           EXIT.
      *
```

```
      * MODULE FUNCTION: THIS MODULE PRINTS OUT EACH RECORD ON THE NEW
      *                  UPDATED RESERVATIONS FILE.
      * CONTROL VARIABLES MODIFIED: RP-END-OF-RESERV
      * MODULES INVOKED:            NONE
      * INVOKING MODULES:           RESERVATIONS-PROGRAM
      *
       RE-PORT SECTION.
       RE-PORT-ENTRY.
           READ NEW-RESERVATIONS-FILE
               AT END MOVE '1' TO RP-END-OF-RESERV
               GO TO RE-PORT-EXIT.
           MOVE NEW-RESERVATIONS-RECORD TO PRT-REC
           WRITE PRINT-RECORD.
       RE-PORT-EXIT.
           EXIT.
```

References

Ball, T. and Eick, S. G. (1994) 'Visualizing program slices', in *Proceedings of the 10th Annual Symposium on Visual Languages*, St. Louis, MO, IEEE Press, Los Alamitos, CA, pp. 288–295.

Biggerstaff, T. J., Mitbander, B. G. and Webster, D. E. (1994) 'Program understanding and the concept assignment problem', *Communications of the ACM*, **37** (5), 72–82.

Chen, Y. F., Nishimoto, M. Y. and Ramamoorthy, C. V. (1990) 'The C information abstraction system', *IEEE Transactions on Software Engineering*, **16** (3), 325–342.

Detienne, F. and Soloway, E. (1990) 'An empirically-derived control structure for the process of program understanding', *International Journal of Man–Machine Studies*, **33** (3), 323–342.

Devanbu, P., Brachman, R. J., Selfridge, P. G. and Ballard, B. W. (1990) 'LaSSIE: a knowledge-based software information system', *Communications of the ACM*, **34** (5), 34–49.

Harandi, M. T. and Ning, J. Q. (1990) 'Knowledge-based program analysis', *IEEE Software*, **7** (1), 74–81.

Hartman, J. (1991) 'Understanding natural programs using proper decomposition', in *Proceedings of the International Conference on Software Engineering*, Austin, TX, IEEE Press, Los Alamitos, CA, pp. 62–73.

Johnson, W. L. (1986) *Intention Based Diagnosis of Novice Programming Errors*, Morgan Kaufman, Los Altos, CA.

Kozaczynski, V. and Ning, J. Q. (1994) 'Automated program understanding by concept recognition', *Automated Software Engineering*, **1** (1), 61–78.

Kozaczynski, V., Ning, J. Q. and Engberts, A. (1992) 'Program concept recognition and transformation', *Transactions on Software Engineering*, **18** (12), 1065–1075.

Kuhu, D. R. (1987) 'A source code analyzer for maintenance', in *Proceedings of the 1987 IEEE Conference on Software Maintenance*, Austin, TX, IEEE Press, Los Alamitos, CA, pp. 176–180.

Letovsky, S. (1988) 'Plan analysis of programs', Ph.D. Thesis, Yale University, New Haven, CO.

Letovsky, S. and Soloway, E. (1986) 'Delocalized plans and program comprehension', *IEEE Software*, **3** (3), 41–48.

Murray, W. R. (1988) *Automatic Program Debugging for Intelligent Tutoring Systems*, Morgan Kaufman, Los Altos, CA.

Ning, J. Q., Engberts, A. and Kozacyzski, W. (1993) 'Recovering reusable components from legacy systems by program segmentation', in *Proceedings of 1993 Working Conference on Reverse Engineering*, Baltimore, MD, IEEE Press, Los Alamitos, CA, pp. 64–72.

Quilici, A. (1994) 'A memory-based approach to recognizing programming plans', *Communications of the ACM*, **37** (5), 84–93.

Quilici, A. and Chin, D. N. (1994) 'A cooperative program understanding environment', in *Proceedings of the Ninth Knowledge-based Software Engineering Conference*, Monterey, CA, pp. 125–132.

Rich, C. and Waters, R. C. (1990) *The Programmer's Apprentice*, Addison Wesley, Reading, MA.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ.

Selfridge, P. G. and Heineman, G. (1994) 'Graphical support for code-level software understanding', in *Proceedings of the Ninth Knowledge-based Software Engineering Conference*, Monterey, CA, IEEE Press, Los Alamitos, CA, pp. 117–123.

Soloway, E. and Erdlich, K. (1984) 'Empirical studies of programming knowledge', *IEEE Transactions on Software Engineering*, **10** (5), 595–609.
Takeda, K., Chin, D. N. and Miyamoto, I. (1992) 'MERA: Meta language for software engineering', in *Proceedings of the 4th International SEKE Conference*, Capri, Italy, IEEE Press, Los Alamitos, CA, pp. 495–502.
Weiser, M. (1982) 'Programmers use slices when debugging', *Communications of the ACM*, **25** (7), 446–452.
Weiser, M. (1984) 'Program slicing', *IEEE Transactions of Software Engineering*, **10** (4), 252–357.
Wilde, N., Huitt, R. and Huitt, S. (1989) 'Dependency analysis tools: reusable components for software maintenance', in *Proceeding of the 1989 Conference on Software Maintenance*, Miami, FL, IEEE Press, Los Alamitos, CA, pp. 126–127.
Wills, L. M. (1990) 'Automated program recognition: a feasibility demonstration', *Artificial Intelligence*, **45** (1–2), 113–172.
Wills, L. M. (1992) 'Automated program recognition by graph parsing', Ph.D. Thesis, Technical Report 1358, MIT Artificial Intelligence Laboratory, Cambridge, MA.
Zimmer, J. A. (1990) 'Restructuring for style', *Software Practice and Experience*, **20** (4), 365–389.

Authors' biographies:

David N. Chin is an Associate Professor of Information and Computer Sciences at the University of Hawaii. He received his B.S. from M.I.T. and his Ph.D. from U.C., Berkeley, where he developed UC, the UNIX Consultant, an intelligent agent that answers questions about the UNIX operating system. His current research interests include artificial intelligence (particularly natural language processing and intelligent agents), software engineering, user modelling, and intelligent interfaces (including geographical information systems).

Alex Quilici is an Associate Professor in the Department of Electrical Engineering at the University of Hawaii at Manoa. He received his A.B. from U.C. Berkeley and his Ph.D. from U.C.L.A. His primary research interests lie in creating intelligent user interfaces and in applying Artificial Intelligence techniques to automate or assist various software engineering tasks, especially software maintenance, module-based software construction.